Towards Fully Automated TLM-to-RTL Property Refinement

Vladimir Herdt¹ Hoang M. Le¹ Daniel Große¹,² Rolf Drechsler¹,²

¹Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
²Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{vherdt,hle,grosse,drechsle}@informatik.uni-bremen.de

Abstract—An ESL design flow starts with a TLM description, which is thoroughly verified and then refined to a RTL description in subsequent steps. The properties used for TLM verification are refined alongside the TLM description to serve as starting point for RTL property checking. However, a manual transformation of properties from TLM to RTL is error prone and time consuming. Therefore, in this paper we propose a fully automated TLM-to-RTL property refinement based on a symbolic analysis of transactors. We demonstrate the applicability of our property refinement approach using a case study.

I. INTRODUCTION

In the recent years, the emergence of the Electronic System Level (ESL) [1] can be witnessed. It has become an industry common practice to model hardware designs at this new level of abstraction using the IEEE standard language SystemC [2], [3]. At ESL, functional behaviors of a design are described using the full descriptive power of C++ together with the added layer of concurrency by SystemC, while its communication interfaces are abstracted to function calls and parameters. The synchronization between functional behaviors and communication interfaces is realized using events. These modeling techniques are termed as Transaction Level Modeling (TLM) and standardized as a set of interfaces as TLM-2.0 [2].

In the top-down ESL design flow, SystemC TLM models are available very early and get successively refined to RTL. TLM models also serve as reference models to verify the corresponding RTL models later. The verification is commonly performed by a TLM/RTL co-simulation, where the same input stimuli are applied to both models and their outputs are checked for equivalence. This step requires an additional executable verification component, known as transactor, to bridge the function calls at TLM with the signal-based interfaces at RTL and vice versa, as shown in the upper half of Fig. [1].

Obviously, the correctness of TLM models is also of great importance. In the past few years, a wide body of verification techniques at TLM has been developed ranging from simulation-based (e.g. [4]–[6]) to formal verification (e.g. [7]–[14]). Please note that we do not try to be comprehensive but only mention a few representative approaches. Most of these approaches are based on the principle of Assertion-Based Verification (ABV) [15], which has been very successful for RTL verification. Assertions, also known as properties, provide a mean to formally capture the functional specification of the design. They specify conditions on design inputs, outputs and internal variables that are supposed to hold at all times. These conditions can also be temporal, i.e. referring to a sequence of values over time. Assertions can be automatically translated into executable monitors to be co-simulated with the design to check the specified conditions. For the specification of TLM properties, extensions to standardized language such as IEEE-1850 PSL or IEEE-1800 SVA have been proposed [4], [16].

While the verification step based on TLM/RTL co-simulation is indispensable, the obtained results are far from complete and cannot ensure the absence of bugs at RTL. For some special cases when the TLM models are high-level synthesizable (e.g. when all used constructs are in the SystemC Synthesizable Subset), formal sequential equivalence checking is supported by existing EDA tools such as Calypto SLEC or Synopsys Hector. For general TLM models, it is highly desirable that co-simulation is complemented with other (preferably formal) approaches. A promising direction is to reuse properties that have been proven on the TLM model. Due to the semantic differences of the involved abstraction levels, straightforward reuse is not possible. The translation process, i.e. TLM-to-RTL property refinement, is mostly manual, therefore error-prone and time-consuming.

Related Work: Several improvements to the manual TLM-to-RTL property refinement have been proposed. Ecker et al. [17] formulated a set of requirements for the refinement process. In a follow-up work [18], an automated refinement framework has been introduced. Still, a set of refinement rules have to be defined by verification engineers before the automated translation can start. Chen and Mishra [19] proposed a similar approach that requires a formal (temporal) semantic mapping between TLM functions and clocked RTL signals. Pierre and Amor [20] developed a set of pre-defined “pattern-matching” rules to ensure that the refined RTL properties belong to the simple subset of PSL, which is generally easier to verify. However, the need for a manual semantic mapping is still not circumvented. Bombieri et al. [21] proposed to reuse TLM properties in a TLM/RTL co-simulation. Thanks to the availability of transactors, RTL signals over time are converted back to TLM transactions and the TLM properties can thus be checked on the RTL design. This is, however, not a semantic approach, i.e. no corresponding RTL properties can be derived to apply, for example, RTL property checking.

Paper Contribution and Organization: To the best of our knowledge, we propose in this paper the first fully automated TLM-to-RTL property refinement approach. The lower half of Fig. [1] shows an overview. The main idea lies in a better reuse of the readily available transactors. At the core of our approach is a static transactor analysis based on symbolic execution (Section [III]). Essentially, the analysis reverse-engineers the executable transactors to create
III. STATIC ANALYSIS OF TRANSACTORS

This section describes our static analysis based on symbolic execution to extract the underlying transactor protocol between TLM and RTL as FSMs.

A. Symbolic Execution

Starting with the initial symbolic execution state, the transactor function is repeatedly executed, until all reachable states are explored. Fig. 4 shows the complete symbolic state space of the transactor. In total we have 7 distinct states (\{S0,...,S6\}), which make up the state space, and 6 additionally observed states (\{∗S7,...,∗S12\}) that are equivalent to one of the first 7 and thus are discarded.

The initial execution state S0 is defined as follows: RTL signals and TLM input are externally provided and therefore initialized with symbolic values. The protocol data (status and index i) are initialized with concrete values. The protocol data (status and index i) are initialized with concrete values. The protocol data (status and index i) are initialized with concrete values. The protocol data (status and index i) are initialized with concrete values. The protocol data (status and index i) are initialized with concrete values. The protocol data (status and index i) are initialized with concrete values.

The initial execution state S0 is defined as follows: RTL signals and TLM input are externally provided and therefore initialized with symbolic values. The protocol data (status and index i) are initialized with concrete values. The protocol data (status and index i) are initialized with concrete values.
IV. PROPERTY REFINEMENT

This section outlines the refinement process based on the extracted FSMs. Before going into details, we briefly describe the used property language as well as the actual TLM properties for refinement.

A. Property Specification Language

For property specification, we consider the simple subset of PSL with TLM extensions similar to [3], [16]. For TLM properties, we use TLM events as atomic boolean propositions. For every function $f$ we define the events $f\text{begin}$ and $f\text{end}$ (abbreviated as $f\text{b}$ and $f\text{e}$). Arguments can be captured to reason about the data, e.g. $in(x);e$ denotes that the function has finished with $x$ being passed as parameter. Finally, $e\text{notified}$ denotes that event $e$ has been notified. Boolean C++ expressions can be embedded to reason about data relations. We use SEREs (sequentialized regular expressions) to specify RTL traces (i.e. signal sequences sampled at clock cycles).

B. TLM Properties for Refinement

We consider two TLM properties P1 and P2, shown in upper part of Table I. P1 specifies that after a DMA read a DMA event notification needs to happen before the next DMA read is started. This ensures that data is available before it is attempted to read. P2 checks that no packets are lost. It captures the input packet in the variable $x$ and when the input is completed, either the next or second next DMA read will read $x$. The $next\_event(\alpha, \phi)$ operator requires that $\phi$ is true at the next occurrence of $\alpha$. The $CEQ(y, x)$ predicate returns true when the DMA result and input cell are equal. The predicate can be expressed in C++ syntax as shown in Table I (upper part).

C. Refinement Process

Our refinement works recursively. The operators $always$, $next$, $before$, $\\|$ and $next\_event(\alpha, \phi)$ are preserved. The implication $L\rightarrow R$ is mapped to $L\\rightarrow R$, i.e. first $L$ needs to be matched and matching of $R$ starts at the clock cycle $L$ has been matched at. This is a valid refinement thanks to the monotonic advancement of time required by the simple subset.

A TLM event $E$ is mapped to a sequence of RTL signals (i.e. traces) by following back all edges starting from FSM state $S$, event $E$ is associated with, and collecting the signal assignments until the resulting traces $T$ uniquely identify $S$. 
TABLE I

<table>
<thead>
<tr>
<th>Function Call Events Capturing Input/Output Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>H1</strong></td>
</tr>
<tr>
<td><strong>H2</strong></td>
</tr>
<tr>
<td><strong>RTL(read;y)</strong></td>
</tr>
<tr>
<td><strong>RTL(read;x)</strong></td>
</tr>
<tr>
<td><strong>RTL(read;e)</strong></td>
</tr>
<tr>
<td><strong>RTL(read;x)(y)</strong></td>
</tr>
<tr>
<td><strong>RTL(read;e)(x)</strong></td>
</tr>
</tbody>
</table>

Upper half shows the TLM properties and equality predicate CEQ. Middle half shows mapping of TLM events to RTL signal sequences, and lower half shows the refined results.

i.e. starting from any state Q in the FSM and following any sequence \( t \in T \), only state S can be reached (otherwise there has been an invalid transition). We call \( T \) the distinguishing traces of state \( S \), and can straightforwardly express \( T \) as a SERE by sequencing (operator \( ; \)) all signal assignments within a trace and then union all traces (operator \( \cup \)). Table [I] (middle part) shows the results for the relevant TLM events. For example, notified (associated with state S6 in output FSM) is uniquely identified by \( done \) (i.e. \( done=1 \)), as S5 is the only state where \( done \) is a valid signal assignment. On the other hand to identify S5 (associated with read:e) going back one step is not enough, as many states can be reached with \( !done \). In fact, \( T \) for S5 contains two sequences of length 4.

Function call events capturing input/output arguments require an extended refinement procedure. It is not enough to just visit the FSM state associated with the function begin/end, but also go through all states where a data mapping to the arguments exists. The event \( in(x):e \) requires to visit the state sequence \([S3,S4,S5]\) in the input FSM. Therefore, we generate \( T \) for the first state in the sequence and extend it with signal assignments to reach the subsequent states by following along the FSM edges. At each state we capture the current data signals in a new local variable for S3, e.g. \( NEW(x_0:=data) \) (see [20]). Based on the data mapping from the FSms, we obtain that RTL signal \( data \) in S3 (input FSM) is mapped to \( buf[0] \) and following along the data mapping of the TLM input function, that \( buf[0] \) maps to \( x.data[0] \) of the input cell. Similarly, we obtain mappings for the output side. With the captured data signals and mappings, we can automatically refine the \( CEQ(y,x) \) predicate (Table [II] lower part).

**V. DISCUSSION AND FUTURE WORK**

Our proposed **fully automated** TLM-to-RTL property refinement approach is still a work-in-progress. The monotonic advancement of time through the formula allows for an intuitive recursive refinement algorithm, where evaluation of subformulas starts at the clock cycle the previous one finished at. Although the feasibility of our refinement approach has been demonstrated on a realistic case study with two representative TLM properties, a formal characterization of supported properties is still to be defined in future work. One limitation is that we assume the next (or SEREs in general) operator is not used to specify an absolute order of TLM events. The reason is that, it is unclear how to automatically refine it (especially when combined with other operators), as there can be (arbitrary) many clock cycles between TLM events at RTL. Moreover, it has been suggested that using the next operator in this way at TLM is not well suited (up to the point of being unsound) due to its restrictiveness [20].

Rather, the **before** and **next_event** operators, which specify precedence, should be used. However, using a single next operator in combination with \( -> \) and **next_event** makes sense to avoid matching the subformula at the current instant (e.g. as used in property P2 in Section IV-C to avoid matching the same read). This issue needs further investigation.

For property refinement, we compute distinguishing traces for FSM states. However, this is not possible for FSms that have multiple state cycles, which accept the same input sequence. For a protocol description such an FSM implies that it is not possible to detect the protocol status by just observing the IO signals, i.e. there is no synchronization or handshaking in place. While we believe it is a reasonable assumption to have distinguishing states, our refinement algorithm can be extended to handle non-distinguishing states as well by: 1) accessing internal variables of the model, or 2) generating additional logic between model and monitor to simulate the FSM (can be done automatically). We plan to evaluate these extensions in future work.

### References


